Machine Learning with the backpropagation algorithm

Author: Aleix Espuña Fontcuberta.
*Facultat de Física, Universitat de Barcelona, Diagonal 645, 08028 Barcelona, Spain.**

Advisor: LLuís Garrido Beltrán

Abstract: The goal of this project was to develop with Python a classifier capable to separate two different classes (binary classifier). To do this, we implemented a neural network that uses gradient descent with the backpropagation algorithm to learn in a supervised training process. We first trained the net and studied its behavior with a binary classification problem that we created, generating our own training examples with the acceptance-rejection algorithm. We realized that the net was returning the conditional probability for each example to belong to one of the classes. After that, we focused on two different real classification problems, both obtained from available online machine learning data sets.

I. INTRODUCTION

Classifying a set of examples according to the classes they belong to is a problem that can be commonly found in physics or in other disciplines. It is often necessary to classify a set of physical measurements where each one can correspond to a signal we want to analyze or to a noisy background we want to discard. For example, if we are studying the trajectory of a subatomic particle such as the Higgs boson, we must first know if the positions we are measuring correspond to the Higgs boson or to any other particles which are also running in our accelerator (background). In general, an example i to classify can be described as a vector $\vec{x}_i$, whose values are the features of that example. A set of examples $\{\vec{x}_1, \vec{x}_2, ...\vec{x}_m\}$ that belong to one class, lets say class 0, can be distributed according to a distribution function $\rho_0(\vec{x})$. The problem is that in many complex classification tasks, the distribution functions of the different classes are very complicated and unknown. There are efficient methods for classifying examples if the analytical distribution functions of the different classes are known, but this is not the case we consider here. When the distribution functions are unknown, Machine Learning techniques can be used to develop a function that learns to classify the examples. In our project, we focused on binary classification tasks. We used a neural network that learns in a supervised training process called gradient descent, where the gradient is calculated with an algorithm named backpropagation [1]. The training is called supervised because every time the net receives a training example, it also receives its solution, which is the class number where the example belongs to (0 or 1). First, explanation about what is a neural network and how the backpropagation algorithm works will be provided. Afterwards, we will show how we used a method called acceptance-rejection to generate training data and build a very instructive classification problem. With it, we will understand which is the ideal function that our net tries to reach during the training process of this problem. Finally, we will study two different real classification problems obtained from online ML data sets, exhibiting the neural network training and performance in each one.

A. Neural networks with the sigmoid function

To understand what is a neural network, it is convenient to take a look at Fig.(1). Each circle represents a neuron, which has an intrinsic parameter that is commonly called the bias b. The connection between two neurons also has an intrinsic parameter, the weight w. The last parameter is the activation a, which represents the information that one neuron sends to the others located at the next layer. When a single neuron in a layer receives M activations from the previous layer of M neurons, a new variable z is created by computing $z = (\sum_{i=1}^{M} a_i w_i) + b$. Finally, the sigmoid function is applied to the z variable, generating $\sigma(z) = \frac{1}{1+\exp(-z)}$, which becomes a new activation that is redirected to all the next layer neurons.

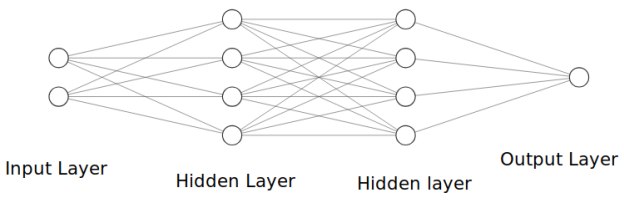

FIG. 1. Example of a binary neural network with four layers.

From now on, we will use the same notation that [1] uses to refer to all parameters found in a layer. Activations a^l and bisases b^l will be vectors. The superscript l will indicate the layer they belong to. To give you just one example from Fig.(1), the first hidden layer has four neurons and therefore the biases form a vector b^2 of size 4. Each neuron in the first hidden layer computes the

* tfgac@ub.edu

sigmoid function and generates an activation, so the first hidden layer creates the activations vector a^2, also of size 4. Weights are more complicated because they will be represented as matrices. A matrix w^l will contain the weights of all connections between the l and $l-1$ layers. In Fig.(1), the value w^2_{ij} of the 4x2 matrix w^2 corresponds to the weight between the i neuron of the second layer and the j neuron of the first layer. With this new vector notation, the result a^l that a layer l redirects is a vector $a^l = \sigma(z^l)$ with $z^l = w^l a^{l-1} + b^l$. Note that $w^l a^{l-1}$ is the product of a matrix for a vector, returning a vector. Note as well that the expression $\sigma(z^l)$ means that the sigmoid function is being applied to each of the z^l components, returning a vector too.

The process for calculating the neural net output given an $\vec{x}_i$ example starts at the input layer. This input layer does not do any computation, it just sends the example to the next layer, which means $\vec{x}_i = a^1$. Then, the described products and sigmoid functions are computed layer by layer until reaching the net output a^L, where L means the last layer. In Fig.(1), $a^L = a^4$ is a vector of size 1, or in other words, a single value. That's why we call the net in Fig.(1) a binary neural network, because it returns a number between 0 and 1 that will be used for binary classification tasks. In this context, learning means trying to achieve a neural network function $a^L(\vec{x})$ that minimizes an error defined as $E = \frac{1}{2N}\sum_x (y(\vec{x}) - a^L(\vec{x}))^2$. Where the sum goes through a group of N examples $\vec{x}$ whose solutions $y(\vec{x}) \in \{0, 1\}$ are known. This examples are called training examples and all together form what we call a training set. It will be useful to rewrite the error in the form $E = \frac{1}{N}\sum_x E_x$, where $E_x = \frac{1}{2}(y(\vec{x}) - a^L(\vec{x}))^2$.

II. THE BACKPROPAGATION ALGORITHM

A. About gradient descent

Gradient descent consists in changing the neural net's weights and biases towards the opposite direction of the E function's gradient in order to reach the minimum of the error E. Even though there are different methods to compute gradient descent, we will focus on the fast approach that Eq.(1) describes, which also appears in [2].

$$\Delta w^l_{ij}\{n\} = -\eta \frac{\partial E_x}{\partial w^l_{ij}} + \alpha \Delta w^l_{ij}\{n-1\} \tag{1}$$

$$\Delta b^l_i\{n\} = -\eta \frac{\partial E_x}{\partial b^l_i} + \alpha \Delta b^l_i\{n-1\}$$

Please notice that the update that a weight or bias receives in a step $\{n\}$ depends on the update it received in the previous step $\{n-1\}$. The most difficult task is to calculate the gradient. Even a very small neural network can easily have hundreds of weights and biases, which means that hundreds of partial derivatives will have to be calculated. This task is carried out with the backpropagation algorithm.

B. The backpropagation equations

The reader can also study the same equations in further detail in chapter 2 of [1]. The backpropagation equations are written in terms of a new variable δ^l. This new variable is defined as a vector $\delta^l \equiv \frac{\partial E}{\partial z^l}$, which means that $\delta^l_j \equiv \frac{\partial E}{\partial z^l_j}$.

$$\delta^L = \nabla_a E \odot \sigma'(z^l) \tag{2}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{3}$$

$$\frac{\partial E}{\partial b^l_j} = \delta^l_j \tag{4}$$

$$\frac{\partial E}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{5}$$

It's easy to understand why this four equations are called backpropagation equations. The delta parameter is first calculated in the last layer with Eq.(2), where the Haddamart product $\odot$ appears. Then, Eq.(3) lets us to recursively calculate the delta variables in the previous layers. Finally, the gradients are calculated with Eq.(4) and Eq.(5).

III. UNDERSTANDING THE NEURAL NETWORK OUTPUT

We wanted to start with a simple binary classification problem. It was based on classifying 2D random points generated according to two simple distribution functions defined by us. In order to generate data for this problem, we considered that we could use a well known algorithm called the acceptance-rejection algorithm, listed in [3]. We first defined the functions $\rho_1(x_1, x_2) = \frac{2}{\log(2)}\frac{x_1}{x_2+1}$ and $\rho_0(x_1, x_2) = 9x_1^2x_2^2$, both normalized in the coordinates domain $x_1, x_2 \in [0, 1]$. After that, we used the algorithm to generate 10^4 2D points, 50% of them distributed according to ρ_0 and the other 50% according to ρ_1. We wrote the coordinates of this points and the class where each one belonged to in a file called training file. We also used the same algorithm to create an additional file, called test file, with the same characteristics. The reason why we created this test file is that you can train very well the net with your training data and yet obtain poor results when trying to classify new examples that your net didn't see during the training. Checking that both the training file error and the test file error are decreasing during the training lets you to control that your

net is performing well on the training file and on new information as well. In this problem, the error function E that we described in the introduction can be approximated in an integral form considering a continuous case of two probability densities. Once it is done, a functional analysis on the new error can be performed, shown in [2], finding the analytical function $a_*^L(x_1, x_2)$ that minimizes the error E.

$$a_*^L(x_1, x_2) = \frac{\alpha_1 \rho_1(x_1, x_2)}{\alpha_1 \rho_1(x_1, x_2) + \alpha_0 \rho_0(x_1, x_2)} \qquad (6)$$

The alpha variables of Eq.(6) are the proportion of points belonging to each distribution. In this case, we have mentioned that $\alpha_0 = \alpha_1 = \frac{1}{2}$. The reader now can realize that Eq.(6) is indeed the Bayes' Theorem applied to our particular problem, it is a conditional probability. More exactly, it is the probability for a point to belong to distribution 1 if we know that it has fallen in a differential region centered at $\vec{x} = (x_1, x_2)$. We can refer to the function $a_*^L(x_1, x_2)$ in Eq.(6) as $P(1 \mid \vec{x})$.

The structure of the neural network that we created with Python to compute this training corresponds to the structure that Fig.(1) shows, which means we used two hidden layers of 4 neurons each one. We took a learning rate $\eta = 0.1$ and a momentum rate $\alpha = 0.5$. We iterated the training file 10000 times, updating the weights and biases according to Eq.(1) after reading each training example. After each training file iteration, we wrote the training file error, the test file error and the neural network weights and biases in three different files. By plotting the training file errors, we checked that the net was indeed trying to reach a minimum error in the training file, see Fig.(2). The reason we also saved the test file errors is to control a phenomenon called overlearning. It occurs when the error calculated in the test file starts to indefinitely grow after each training file iteration, meaning that too many iterations have been made and that the net is not generalizing, it is not classifying well new information.

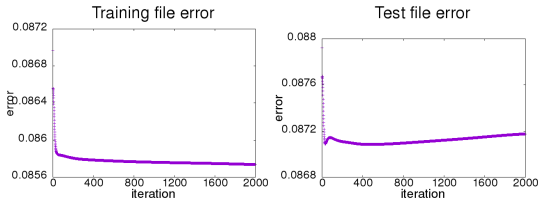

FIG. 2. Sows the evolution of the training and test errors after each training file iteration. The training error is clearly being minimized but overlearning occurs in the test file after iteration 472, which gives the minimum test error. A local minimum is also observed in the test file error plot, located before iteration 200.

After the iterations, we searched the iteration that was giving the lowest error in the test file and chose the corresponding neural net weights and biases as the best parameters. With them, we evaluated the neural network at each point of the test file. Fig.(3) shows that indeed the neural network output function that we obtained is quite similar to the one that Eq.(6) predicted. The main difference can be observed in the separation curve of probability 0.5 that each plot shows. The reason why both curves are not the same is because our neural network is not big enough. The bigger the number of layers and neurons of a neural network is, the bigger the similarity between both figures should be.

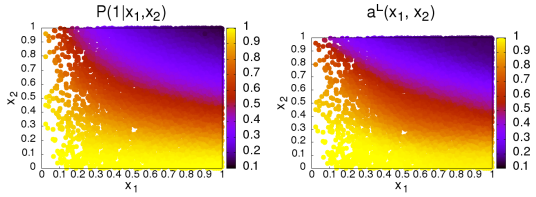

FIG. 3. Comparison between the analytical function $P(1 \mid \vec{x})$ and the neural net function $a^L(x_1, x_2)$, both evaluated at the test file points.

So until now we still haven't classified anything, as at the moment we only have a function closer to the conditional probability $P(1 \mid \vec{x})$ for each point. In order to do the classification task, a threshold C must be chosen such that if $a^L(\vec{x}) > C$ the point will be classified as 1, or classified as 0 if $a^L(\vec{x}) < C$. Knowing that, we defined the efficiency of a threshold value C as the number of correct classifications made in the test file with it. With this definition, the theoretical optimal threshold C for the function $P(1 \mid \vec{x})$ is the intuitive value $C = \frac{1}{2}$. In our case, with the trained function $a^L(\vec{x})$, we found that the best threshold for the test file was $C^* \approx 0.505$, with an efficiency of about 74%, as Fig.(4) shows.

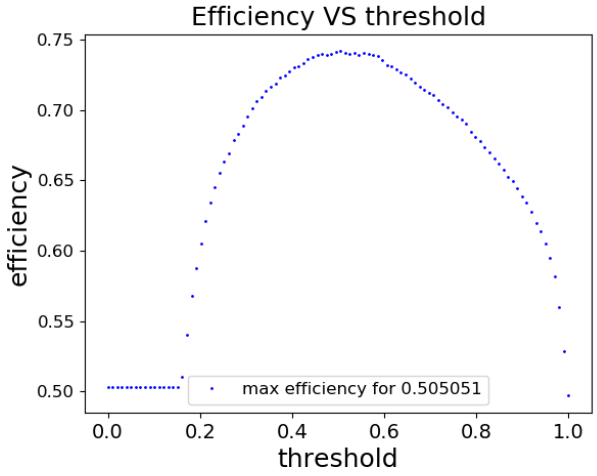

FIG. 4. Classification of the test file points with the final function $a^L(x_1, x_2)$ for 100 different thresholds. The results agree with theory, as the maximum efficiency of 74% is obtained with a threshold value very close to 0.5.

There is one last thing worth to mention about the

theoretical optimal threshold, which can also be found in [2]. If the training file proportions α_0, α_1 are not the same than the test file proportions α'_0, α'_1, then the conditional probability of the training file points given by Eq.(6) is not the same than the conditional probability of the test file points, which now is given by the function $P_{test}(1 \mid \vec{x}) = \frac{\alpha'_1\rho_1(x_1,x_2)}{\alpha'_1\rho_1(x_1,x_2)+\alpha'_0\rho_0(x_1,x_2)}$. When that happens, the new optimal threshold is $\beta = \frac{1}{1+\frac{\alpha_0\alpha'_1}{\alpha'_0\alpha_1}}$.

IV. THE MAGIC GAMMA PROJECT

The Magic Gamma is the first real classification problem we focused on. The data set was obtained from an online ML repository, the reader can find the web page link in [4]. The Magic Gamma was a project in which collaborated more than 20 universities from all over the world. You can visit [5] to obtain more information about the physics and the telescopes involved in the research. MAGIC means Major Atmospheric Gamma Imaging Cherenkov. The goal of the project was to study high energy gamma rays entering the atmosphere. According to [5], when one of this gamma rays enters the atmosphere, it produces a shower of relativistic charged particles, which are mainly electrons and positrons. After that, the charged particles radiate, emitting in a cone a special radiation called Cherenkov radiation. This Cherenkov radiation is finally detected by the telescopes, forming an elliptical image on a camera plane. Information about the initial gamma rays can be obtained by studying the ellipses formed by the Cherenkov radiation. The main problem is that many other much frequent cosmic rays like protons or light ionized atoms can also produce hadronic showers and the Cherenkov radiation. Consequently, it is necessary to distinguish the ellipses generated by the interesting gamma rays from the ellipses generated by some hadrons (background). We can teach our neural network to do that by using the available data that [3] provides. This training data was generated in a very sophisticated Monte Carlo simulation that we will not explain here, full details are found in [6]. What matters to us is that the repository provides us a file of 19020 lines with 11 columns. Each line contains 10 different characteristics of the image of a particular ellipse and its solution (gamma or hadron). This 10 attributes are described in [4], they are the length of the major and minor axis of the ellipse, the sum of the pixels of the observed image, etc. In order to obtain better results, it is advisable to normalize the values of each column to numbers between 0 and 1. By doing that, the neural network does not receive data which has big fluctuations. If c_i^1 is the value found in line i of the first column, we normalized that value by doing $c_i^1 = \frac{c_i^1 - min(c^1)}{max(c^1)-min(c^1)}$, where $max(c^1)$ is the maximum value found in c^1 and $min(c^1)$ is the minimum one. Finally, we divided the data file in two files, the training file and the test file, and proceeded like in the previous problem. The structure of the hidden layers (4x4) and the learning rates where the not changed ($\eta = 0.1$, $\alpha = 0.5$). The proportions of the training and the test files were almost the same ($\alpha_1 = 0.6479, \alpha'_1 = 0.6487$), giving a theoretical optimal threshold $\beta = 0.4990$. The empirical results were an efficiency of 87% for a threshold $C^* \approx 0.447$, see Fig.(5) and Fig.(6).

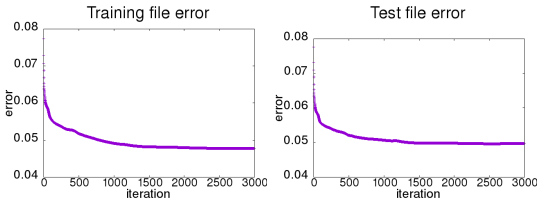

FIG. 5. The evolution of the error in both files is similar, it shows that at the beginning the net is learning to classify training data and new information as well. The lowest error in the test file was found in iteration 2564.

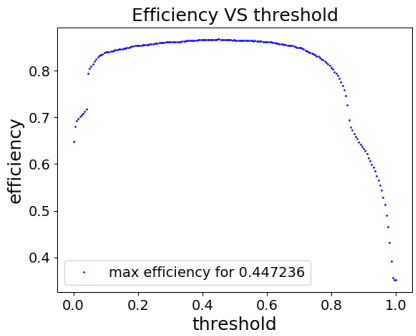

FIG. 6. The efficiency in the test file for 200 different thresholds. The threshold 0.447 gives a maximum efficiency of about 87%.

V. THE IONOSPHERE DATA SET

This data set has also been downloaded and understood from [4]. The data was collected by the Space Physics Group of The Johns Hopkins University Applied Physics Laboratory. The goal was to study the ionosphere in the range of 100 to 500 km altitude. To do that, they used a radar and signal processing techniques. A good summary of the project can be found in [7]. The main idea is that the ionosphere contains free electrons, which can be targeted by multi-pulse waves emitted by a radar. From the study of the back-scattered signals that return from the ionosphere, physical properties of this layer can be derived, like the velocity of the free electrons. In this project, machine learning was needed because sometimes some of the received signals are not suitable for studying the ionosphere. This can happen for different reasons, for example because the signal hasn't

interacted with the ionosphere or because it has suffered several interferences or reflections, which result in phase cancellations and absorptions. When that happens, the signal must be discarded from the set of observations because information about the ionosphere can't be extracted from it. Before the appearance of ML, this demanding classification task required human intervention, which created the training data that we have used. The ionosphere data set consists of the signal description of 351 returned pulses, each one labeled with the letter "g" or "b", meaning good for studying the ionosphere or bad for it. Each received pulse is described by seventeen time sampled measures of its intensity $C(t)$: $\{C(t), C(t+T), C(t+2T), ...C(t+16T)\}$. Where T is the period of the pulse and t is an instant of time. Each training example $\vec{x}_i$ describes a pulse and therefore contains this 17 intensities. However, it is mentioned in [7] that $C(t)$ is a complex number, which means that the final size of a training example $\vec{x}_i$ will be 17*2=34, as we can only store real values in the neural network. To start the training process, once again we split the data in two files, the training file and the test file. This time the class 1 proportions were $\alpha_1 = 0.625$ and $\alpha_1' = 0.657$, giving an expected optimal threshold $\beta = 0.465$. We used the same learning parameters and the same structure of hidden layers. We found $C^* \approx 0.384$, far from β but with an excellent efficiency of 90%, it is showed in Fig.(8).

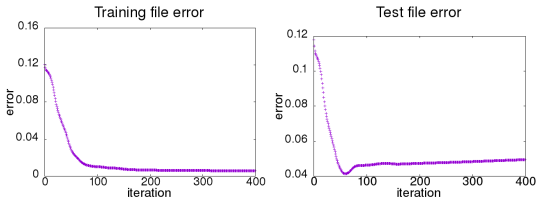

FIG. 7. The training file error of the ionosphere data evolves as expected and the test file error suffers overlearning quite early, which is something typical of small sized training files.

VI. CONCLUSION

We have been able to minimize the training error in three different cases, clearly observing that gradient descent works and obtaining high efficiencies in the last two projects. We have learnt that when solving ML problems under unknown and complicated distributions ρ_0 and ρ_1, one can not achieve with a small net the ideal output function described by Eq.(6). Moreover, we have also seen that an insufficient amount of training data is another important factor that leads to early overlearnings and results distant from the theoretical ones. When solving ML problems under these conditions, one must choose the empirical threshold that gives the best efficiency in the test file, without being worried if it's equal to the theoretical one or not. The real important thing that must be controlled during a training process is the

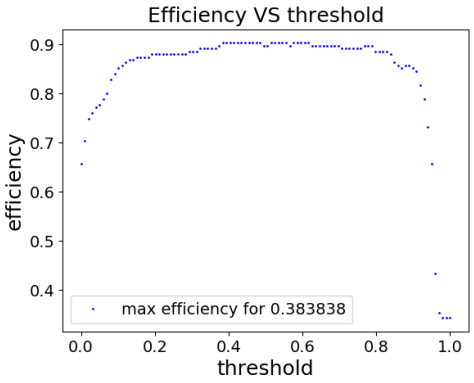

FIG. 8. Classification of radar returns is a problem designed for neural networks, with this simple net of two small hidden layers the efficiency is about 90%.

error of the training and the test files, ensuring that the net is learning to classify the training file and also new information contained in a separate file, the test file.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Lluís Garrido Beltrán for introducing me to the fascinating field of machine learning. His constant advice and supervision gave me a priceless insight into the basics of machine learning. I would also like to show gratitude to my family for their constant support and guidance since I started the physics degree.

[1] M. A. Nielsen, *Neural Networks and Deep Learning* (Determination Press, 2015).
[2] L. Garrido and V. Gaitan, International Journal of Neural Systems **02**, 221–228 (1991).
[3] R. D. Peng, "6.3 rejection sampling," in *Advanced Statistical Computing*.
[4] D. Dua and C. Graff, "UCI machine learning repository," (2017), find the Magic Gamma in: `https://archive.ics.uci.edu/ml/datasets/magic+gamma+telescope`. Find the ionosphere in: `https://archive.ics.uci.edu/ml/datasets/ionosphere`.
[5] "Cherenkov telescopes: Detection of air showers," Find it in `https://magic.mpp.mpg.de/newcomers/cherenkov-telescopes/air-showers/`.
[6] D. Heck, J. Knapp, J. N. Capdevielle, G. Schatz, and T. Thouw, (1998).
[7] V. G. Sigillito, S. P. Wing, L. V. Hutton, and K. B. Baker, Johns Hopkins APL Tech. Dig **10**, 262 (1989).